Taller Guiado - Análisis de Algoritmos

Alberto Luis Vigna Arroyo, Camilo José Martinez

4 de febrero de 2024

Abstract

En este documento se presenta el análisis del algoritmo de ordenamiento Bubble Sort, Insertion Sort y Quick Sort. Cada uno de estos algoritmos aborda el problema de ordenar una lista de elementos, pero difieren en sus enfoques y eficiencias. A continuación, se proporciona un resumen de los aspectos más destacados de cada algoritmo, destacando sus características, ventajas y desventajas, así como sus complejidades temporales y espaciales. Este análisis permite comprender mejor la idoneidad y el rendimiento de cada algoritmo en diferentes contextos y escenarios.

Part I
Análisis y diseño del problema

1 Análisis

El problema, informalmente, se puede describir como calcular para una secuencia de elementos $S = \langle a_1, a_2, a_3, \cdots, a_n \rangle$ en donde $\forall_i a_i \in \mathbb{T}$ y $\mathbb{T}$ existe una relación de orden parcial $' \leq'$, una secuencia ordenada, es decir una permutación $S = \langle a'_1, a'_2, a'_3, \cdots, a'_n \rangle$ en donde $a_i \leq a'_{i+1}$ y $a'_i \in S$.

2 Diseño

Con las observaciones presentadas en el análisis anterior, podemos escribir el diseño de un algoritmo que solucione el problema . A veces este diseño se conoce como el «contrato» del algoritmmos o las «precondiciones» y «poscondiciones» del algoritmo. El diseño se compone de entradas y salidas:

Definition. Entradas:

1. Definición entrada 1
2. Definición entrada 2

Definición. Salidas:

1. Definición salida 1
2. Definición salida 2

Parte II
Algoritmos

3 Opción algoritmo 1 - BubbleSort

3.1 Algoritmo

Este algoritmo se basa en el principio de comparar pares de elementos adyacentes en una lista y realizar intercambios si es necesario, repitiendo este proceso hasta que la lista esté completamente ordenada. En cada iteración se va recorriendo la lista y verificando si el elementos $a_i > a_{i+1}$. Si este lo es los elementos se "rotan" para que estos vayan siendo ordenados.

Algorithm 1 BubbleSort

```
1:  procedure BubbleSort(S)
2:      n ← |S|
3:      for i ← 1 to n do
4:          for j ← 1 to n − i do
5:              if S[j] > S[j + 1] then
6:                  variableTemporal = S[j]
7:                  S[j] = S[j + 1]
8:                  S[j + 1] = variableTemporal
9:              end if
10:         end for
11:     end for
12: end procedure
```

3.2 Complejidad

El algoritmo Bubble Sort tiene órden de complejidad $O(n^2)$. Esto se debe a que emplea dos bucles anidados, donde el bucle externo realiza n iteraciones, y el bucle interno realiza comparaciones y posibles intercambios adyacentes. Esta estructura cuadrática resulta en un rendimiento menos eficiente, especialmente para conjuntos de datos extensos.

3.3 Invariante

En cada iteración del bucle externo, el elemento más grande entre los elementos no ordenados está ubicado en la última posición correcta de la lista.

- **Inicio**: Antes de iniciar el ordenamiento consideramos que la parte derecha de la lista (desde el índice i hasta n) contiene los elementos más grandes y ya está ordenada. Inicialmente, $i = 1$, por lo que la lista completa se toma como no ordenada.

- **Avance**: En cada iteración del bucle externo, el bucle interno compara y posiblemente intercambia elementos adyacentes. Esta operación mueve el elemento más grande a la posición $n - i$, manteniendo la parte derecha de la lista ordenada. El valor de i se incrementa después de cada iteración del bucle externo. Si no hay intercambios durante una iteración completa del bucle externo, significa que la lista está completamente ordenada y el algoritmo se detiene.

- **Terminación:** El proceso continua hasta que $i = n$, momentos en el cual se ha completado el ordenamiento de la lista.

3.4 Notas de implementación

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras porta facilisis ultricies. Donec in posuere lacus. Maecenas volutpat augue ac tincidunt accumsan. In in rhoncus mi, sodales finibus augue. Nulla tristique in est nec elementum. Sed purus sem, lacinia eget turpis ut, dapibus rutrum felis. Sed efficitur venenatis mauris at gravida. Nulla finibus rutrum elit eu rutrum. Curabitur a felis libero.

Praesent lacus est, finibus et elementum nec, convallis sed est. Cras fermentum laoreet eros sagittis fermentum. Donec id augue eget ante pellentesque feugiat et et nisi. Ut ornare feugiat tempus. Sed iaculis eros nec libero ultrices, vel efficitur tortor imperdiet. Aliquam tristique, sapien a commodo rhoncus, neque leo dapibus massa, ac ultrices felis tortor in ex. Donec ac risus libero. Fusce rhoncus diam leo, ornare pellentesque lectus sodales vel. Proin scelerisque porttitor elit, nec suscipit erat vestibulum eget. Proin hendrerit, arcu eget faucibus pharetra, urna nisi euismod quam, ac maximus lacus urna in ante. Etiam sit amet semper augue. Donec in ultrices ipsum, tristique maximus odio. Proin volutpat, quam mattis dictum lobortis, lorem velit rhoncus metus, sit amet iaculis tellus ligula quis est. Fusce interdum eros ex, et feugiat est euismod et. Cras non efficitur neque.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras porta facilisis ultricies. Donec in posuere lacus. Maecenas volutpat augue ac tincidunt accumsan. In in rhoncus mi, sodales finibus augue. Nulla tristique in est nec elementum. Sed purus sem, lacinia eget turpis ut, dapibus rutrum felis. Sed efficitur venenatis mauris at gravida. Nulla finibus rutrum elit eu rutrum. Curabitur a felis libero.

4 Opción algoritmo 2 - Insertion Sort

4.1 Algoritmo

Este algoritmo se basa en el principio de iterar sobre una lista S de n elementos, donde $n = |S|$. La idea es que este algorimo empiece desde el segundo elemento ($i = 2$) y que en cada iteración se compare $S[i]$ con los elementos anteriores (es decir, $S[i-1], S[i-2], \ldots$), desplazándolos hacia la derecha mientras $S[i]$ sea menor. La meta es insertar $S[i]$ en la posición correcta, asegurando que la sublista $S[1:i]$ esté ordenada después de cada iteración.

Algorithm 2 InsertionSort

```
1:  procedure InsertionSort(S)
2:      n ← |S|
3:      for i ← 2 to n do
4:          key = S[i]
5:          j = i − 1
6:          while j > 0 and S[j] > key do
7:              S[j + 1] = S[j]
8:              j = j + 1
9:          end while
10:         S[j + 1] = key
11:     end for
12: end procedure
```

4.2 Complejidad

La complejidad del algoritmos de Insertion Sort es de $O(n^2)$, lo que significa que el tiempo de ejecución crece cuadráticamente con el tamaño de la lista. En el peor caso, donde la lista está en orden descendente, cada elemento requiere comparaciones y desplazamientos en cada iteración, resultando en aproximadamente $\frac{n^2}{2}$ operaciones. Aunque es eficiente para listas pequeñas o casi ordenadas, Insertion Sort puede volverse ineficiente para conjuntos de datos más grandes debido a su complejidad cuadrática.

4.3 Invariante

Después de cada iteración del bucle externo, la sublista que va desde el inicio de la lista hasta el índice actual $S[1:i]$ está ordenada. Esto implica que los elementos en la sublista ya están en sus posiciones finales correctas en el contexto de la lista completa. El algoritmo, en cada paso, inserta el elemento actual en su posición correcta, desplazando los elementos mayores a la derecha para mantener la ordenación.

- **Inicio**: Antes de iniciar el bucle externo, la sublista inicial $S[1:1]$ consiste en un solo elemento, y por lo tanto, está trivialmente ordenada. El

invariante se cumple inicialmente, ya que no hay elementos previos para comparar.

- **Avance**: Durante cada iteración del bucle externo, el algoritmo selecciona un nuevo elemento $S[i]$y lo compara con los elementos anteriores (es decir, $S\left[i-1\right], S\left[i-2\right], \ldots$). El bucle interno desplaza los elementos mayores hacia la derecha, creando espacio para insertar $S[i]$ en su posición correcta. Al finalizar la iteración, la sublista $S[1:i]$ está ordenada, cumpliendo así el invariante. El avance garantiza que, después de cada iteración, la porción de la lista considerada hasta el momento sigue estando en orden.

- **Terminación:** Cuando el bucle externo ha recorrido toda la lista, la sublista completa $S[1:n]$ está ordenada. El invariante se mantiene hasta el final del proceso, asegurando que, en cada paso del bucle externo, la sublista parcial se mantenga ordenada. Por lo tanto, al terminar, el invariante garantiza que la lista completa esté ordenada.

4.4 Notas de implementación

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras porta facilisis ultricies. Donec in posuere lacus. Maecenas volutpat augue ac tincidunt accumsan. In in rhoncus mi, sodales finibus augue. Nulla tristique in est nec elementum. Sed purus sem, lacinia eget turpis ut, dapibus rutrum felis. Sed efficitur venenatis mauris at gravida. Nulla finibus rutrum elit eu rutrum. Curabitur a felis libero.

Praesent lacus est, finibus et elementum nec, convallis sed est. Cras fermentum laoreet eros sagittis fermentum. Donec id augue eget ante pellentesque feugiat et et nisi. Ut ornare feugiat tempus. Sed iaculis eros nec libero ultrices, vel efficitur tortor imperdiet. Aliquam tristique, sapien a commodo rhoncus, neque leo dapibus massa, ac ultrices felis tortor in ex. Donec ac risus libero. Fusce rhoncus diam leo, ornare pellentesque lectus sodales vel. Proin scelerisque porttitor elit, nec suscipit erat vestibulum eget. Proin hendrerit, arcu eget faucibus pharetra, urna nisi euismod quam, ac maximus lacus urna in ante. Etiam sit amet semper augue. Donec in ultrices ipsum, tristique maximus odio. Proin volutpat, quam mattis dictum lobortis, lorem velit rhoncus metus, sit amet iaculis tellus ligula quis est. Fusce interdum eros ex, et feugiat est euismod et. Cras non efficitur neque.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras porta facilisis ultricies. Donec in posuere lacus. Maecenas volutpat augue ac tincidunt accumsan. In in rhoncus mi, sodales finibus augue. Nulla tristique in est nec elementum. Sed purus sem, lacinia eget turpis ut, dapibus rutrum felis. Sed efficitur venenatis mauris at gravida. Nulla finibus rutrum elit eu rutrum. Curabitur a felis libero.

5 Opción algoritmo 3 - Quick Sort

5.1 Algortimo

El algoritmo de QuickSort consiste en organizar una lista S. Inicialmente el algoritmo selecciona un elementos comúnmente llamado "*pivot*" de la lista, particionando los demás elementos en dos sublistas: aquella que contiene los elementos que son menores al pivot ($S_1 = \{x \in S | x < pivot\}$) y aquellos elementos que son mayores ($S_1 = \{x \in S | x > pivot\}$). Este enfoque divide la tarea en subproblemas más pequeños y se combina para lograr la ordenación completa. Se aplica el mismo proceso de manera recursiva a ambas sublistas, seleccionando nuevos pivotes y dividiéndolas hasta que toda la lista esté ordenada. El éxito de QuickSort radica en la elección eficiente del pivot y la partición de la lista en torno a él, reduciendo así el problema original en subproblemas más pequeños.

El paso clave es la elección del pivot y la partición de la lista en torno a él. Un pivot bien elegido mejora el rendimiento del algoritmo. QuickSort es eficiente en la práctica y a menudo supera a otros algoritmos de ordenamiento como Merge Sort debido a su menor número de intercambios.

Algorithm 3 QuickSort

```
1:  procedure QuickSort(S, inicio, fin)
2:      if inicio < fin then
3:          pivotIndex = particion(S, inicio, fin)
4:
5:          QuickSort(S, inicio, pivotIndex − 1)
6:          QuickSort(S, pivotIndex + 1, fin)
7:      end if
8:  end procedure
9:  procedure particion(S, inicio, fin)
10:     pivot = S[fin]
11:     pivotIndex = inicio − 1
12:     for i ← inicio to fin − 1 do
13:         if S[i] ≤ pivot then
14:             pivotIndex = pivotIndex + 1
15:             intercambio(S, pivotIndex, i)
16:         end if
17:     end for
18:     intercambio(S, pivotIndex + 1, fin)
19:     return(pivotIndex + 1)
20: end procedure
21: procedure intercambiar(S, i, j)
22:     temp = S[i]
23:     S[i] = S[j]
24:     S[j] = temp
25: end procedure
```

Parte III

Comparación de los algoritmos